A method for selecting environments for software compatibility testing

Łukasz Pobereżnik
AGH University of Science and Technology, Cracow, Poland

***Abstract*—Modern software is developed to work with multiple software and hardware architectures, to cooperate with various peer components and can be installed in many different configurations. In order to test it, all possible working environments needs to be created. This requires software and hardware resources like servers, networks and software licenses and most important: man-hours of qualified engineers that will have to configure and maintain them. Because resources are usually limited we have to choose a set of configurations with highest impact on quality of software under test. In this paper we present a method of measuring effectiveness of given software environment for discovering defects in software by introducing environment sensitivity measure. We also show how it can be used in simple algorithm used to select best configurations by using only a selected subset of them and progressively modifying it thougout software development process.**

I. Introduction and problem description

SOFTWARE usually does not work alone. It must have an environment that it works in. This environment can be composed from many components like: servers, operating systems, databases, remote services etc. Those components can also have other components that they rely on. Eg. database might need an operating system to work on. Those dependencies create a Component Dependency Graph (CDG) that describes an environment for Software under Test (SUT). Example of such graph is given on Figure 1. This graph shows only general structure of environment. Each component may also have a set of properties like type, version number, architecture type, permissions, locales etc.

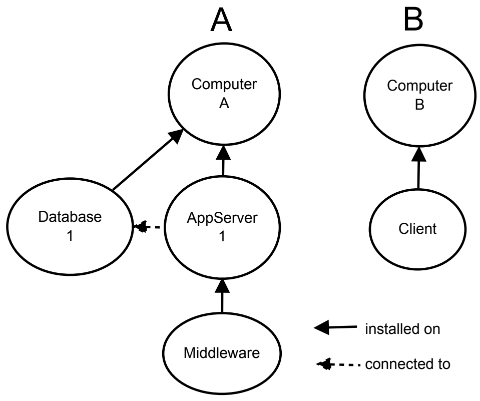

Fig. 1: Example of Component Dependency Graph (CDG) that shows dependencies between resources. Tree A represents environment for server application. Tree B is environment for client application.

Lets take a simple use case: an application working on two operating systems, with three database servers and two application servers. It will give about $2 \times 3 \times 2$=12 different environments to test. If we add another variable: 32-bit or 64-bit architecture, it will double possible environment configurations to at most 24. Adding new configurable element to environment tends to increase the number of possible setups exponentially. Not all configurations may be possible to create (for example some middle-ware may not be available for all operating systems), but it still is significant number of variants to test. Problems of generating test environments and possible solutions were mentioned in our other article [1].

There has been efforts to automate the process of creating those environments based on semantic description of CDG in [2] and [3]. Authors of those articles proposed to use virtualization to construct environments and then use snapshots to clone and then modify them to build other environments. This technique and additional simplifications allowed to reduce number of separate configurations from about 1200 to 160. However this is still to many environments to be build and maintained for everyday regressions tests or continuous builds.

In dissertation [4] same author came also to this conclusion and proposed a manual way to select subset of configurations based on testers' preferences. Decision on which configurations test software is in that case solely based on testers expert knowledge, without support of any analytical tools. In our research we tried to establish a method to measure how good is given configuration for testing and an algorithm to choose the best of them.

II. Selecting best environments for testing

As shown above number of possible environments can be quite high. This means that with limited resources we can only choose subset of them. One of the most popular methods is to use configurations that are most widely used by customers. However when number of software users is high, diversity of configurations may also be too high on and must also be limited.

We have to define what means that one configuration is better for testing purposes than the other and then create an algorithm for choosing the best of them. In our research we followed a common phenomena observed by testers: some of the software environments are causing more problems than the others - basically they fail more tests (or fail them more often). If configuration A is more problematic than configuration B that usually means that if we run tests on configuration A and they will pass, so they will pass also on configuration B (with high probability). This means that we do not need to conducts tests on configuration B so frequently as on configuration A. Conclusion is that configuration A is better for testing than configuration B, because it allows to detect more environment related defects. In order to compare environments it is good to have a numerical metric that will allow to evaluate effectiveness of given configuration. It is also a requirement for many optimizing algorithms (especially evolutionary) to provide a fitness function to compare solutions.

A. Measure of environment sensitivity

In order to compare two environments for software testing we need to establish metric that would tell which configuration is better. Let a T be a set of n tests (test suite) consisting single tests t_i. Let T_k be a vector of test results executed in k iteration. Test can be either 1 (pass) or 0 (fail).

$$T_k = (t_1, t_2, t_3 ... t_n), t_i = 1 \vee 0 \tag{1}$$

E_j is an environment j. Testing function F_T is a function that assigns for each k iteration a vector of tests results T_k to environment C_j.

$$F_T(k, E_j) = (t_1, t_2, t_3, ..., t_n) \tag{2}$$

We can describe F_n for single iteration k in more convenient way as a matrix, where columns are tests and rows are environments (lets note number of configurations as m). t_{ji} is a result of test t_i on environment C_j.

$$F_T(k) = \begin{vmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{vmatrix} \tag{3}$$

First step in calculating sensitivity is to remove those tests that does not bring any information about environment differences. We remove those columns that satisfy condition:

$$\exists p \in [1, m] \forall x \in [1, n] \forall y \in [1, n] : t_{xp} = t_{yp} \tag{4}$$

This means that removed are only those tests that passed or failed in all configurations (remove columns of all 1 or all 0).

Then for each row vector we calculate how many times given test failed and normalize it by number of tests in vector (after removing some of them in first step).

$$Sens(C_j) = \frac{\sum_{x=1}^{n} (1 - t_{jx})}{n} \tag{5}$$

$$Sens(C_j) \in [0, 1] \tag{6}$$

Sensitivity value close to 0 means that given environment is not good for finding defects because all tests here pass. When sensitivity is 1 means that configuration is a good candidate for finding software errors because all tests fail on it whereas on at least one other configuration they pass. Of course, if all tests fail on given configuration we have to check if the problem is not with tests itself - for example there is a defect in testing code.

B. Properties of environment sensitivity

Let's define environment domination: environment A dominates environment B if:

$$\exists x : t_{Ax} < t_{Bx} \wedge \forall y \neq x : t_{Ay} \leq t_{By} \tag{7}$$

In other word: there is at least one test that failed on configuration A but passed on configuration B. This would mean that configuration A found a defect that was not discovered by configuration B.

Environment sensitivity has this property that:

$$A\ dominates\ B \Rightarrow Sens(C_A) > Sens(C_B) \tag{8}$$

This property is result of environment sensitivity definition. Let sum inequalities in second part of domination definition:

$$\sum_{k=1 \wedge k \neq i}^{n} t_{Ak} \leq \sum_{k=1 \wedge k \neq i}^{n} t_{Bk} \tag{9}$$

If we add $t_{Ai} < t_{Bi}$, weak inequality will become strong inequality:

$$\sum_{k=1}^{n} t_{Ak} < \sum_{k=1}^{n} t_{Bk} \tag{10}$$

Note that sensivity calculation requires removing tests that in every configuration failed or passed. This will also convert weak inequality into strong one. If we multiply both sides by -1 and add n:

$$n - \sum_{k=1}^{n} t_{Ak} > n - \sum_{k=1}^{n} t_{Bk} \tag{11}$$

Because $n = \sum_{k=1}^{n} 1$ then we can rewrite equation as:

$$\sum_{k=1}^{n} 1 - \sum_{k=1}^{n} t_{Ak} > \sum_{k=1}^{n} 1 - \sum_{k=1}^{n} t_{Bk} \tag{12}$$

$$\sum_{k=1}^{n} (1 - t_{Ak}) > \sum_{k=1}^{n} (1 - t_{Bk}) \tag{13}$$

Now divide both sides by n:

$$\frac{\sum_{k=1}^{n} (1 - t_{Ak})}{n} > \frac{\sum_{k=1}^{n} (1 - t_{Bk})}{n} \tag{14}$$

And now using environment sensitivity definition:

$$Sens(C_A) > Sens(C_B) \tag{15}$$

Introduction of environment domination allows to us to use existing multi-criteria optimization techniques to find Pareto-efficient solutions. This proof can also be used to quickly compare results of tests run on two configurations without calculating sensitivity itself. Of course it will only introduce order to the set of configurations but would not give any idea how much they differ.

C. Algorithm to generate configurations

Algorithm that will generate and evaluate environments must have several important properties:

1) Works on discrete solution spaces.
2) An ability to search unknown solution space (we have no additional information about local optima) .
3) Iterative schema of work, similar to iterative test execution.

Generating new environment and maintaining it is a costly operation. This means that algorithm must work with small data sets - typically 8-12 configurations. Tests should be run frequently, every code change or at least daily. This means that we may have enough iterations until we reach optimal solution. However we have to remember that each iteration means adding new configuration and this is a costly operation.

Algorithm 1 Simple algorithm for selecting most sensitive environments.

```
$E \Leftarrow GenerateAvailableEnvironments()$ {initial configuration pool}
$P \Leftarrow RandomSubset(E, n)$ {$P$ represents current working set}
$P' \Leftarrow \emptyset$ {$P'$ represents next set}
repeat
  $P'' \Leftarrow P'$ {$P''$ is set from previous iteration}
  $E \Leftarrow E - P$
  $R \Leftarrow RunTests(P)$
  $S \Leftarrow CalculateSensivity(R)$
  $S \Leftarrow SortBySensivity(S)$
  $P' \Leftarrow \emptyset$
  for $i = 0$ to $k$ do
    $P' \Leftarrow P' \cup S[i]$
  end for
  $P \Leftarrow P' \cup RandomSubset(E, n - k)$
until $E$ not empty and $P' \neq P''$
```

$GenerateAvailableEnvironments()$ creates a set of all possible environments we want to execute tests on. Function $RandomSubset(X, n)$ generates a random subset from set X of size n. Summarizing algorithm above: in each iteration we run test suite on n selected environments (working set). Using test results we calculate sensitivity for each of them. After that we select k best of them. From initial pool of environments we choose random ones to fill up working set so it will have n configurations again. Procedure is repeated until all configurations are used (initial pool is empty) or in next two consecutive iterations k best configurations is the same.

This algorithm tends to go though different solutions until it converge to optimal one. For environment compatibility testing this means that before we reach optimal set of configurations we will test much more of them and there is a possibility that we will find even more software defects, than using optimal set from the beginning.

III. Experiments

A. Direct application of environment sensitivity measure

To verify properties of environment sensitivity an experiment was conducted. We used a simple configuration with one operating system and a web browser installed on it. Operating systems used were Linux, Windows and Mac OS X. Browsers under test were Internet Explorer, Firefox, Chrome, Safari, Opera. Each browser was available in several different versions (depending on browser type). System used for experiment was built using small web server (Jetty) that was running a static web page. This web page was based on a popular HTML5 compatibility test site (www.html5test.com) and server both as test suite and application under test. Existing scripts were modified to send test results to data collection servlet running on the same web server (originally they were displayed on the screen). Schematic diagram of system used for experiment is shown on Figure 2.

When the test page was loaded it executed 242 true/false tests and sent results using JSON format back to server where they were stored in file along with information about browser and operating system type. The same page was run on each environment and test results were sent using separate script back to server that stored them for further analysis. Different configurations were provided by web browser compatibility testing cloud service (browsershots.org). We collected results for 46 different configurations. Expected number of environments should be higher, because some of the configurations has been not executed at all by the cloud (which is a defect in cloud service). However number of collected data is enough for analysis. In real life testing setups there are usually no more than several environments in constant use.

Sensitivity measure by definition is calculated relatively to other environments in tested configuration set. In most cases there is not enough resources (computing power, time, machines) to perform tests (and calculate sensitivity) for every environment in given configuration space. We wanted to check how much sensitivity measure will differ when it is calculated for small subset of configuration space against full configuration space. From all 46 environments we chosen randomly subsets of 5, 8 and 10 environments and calculated sensitivity for each configuration in it. We observed that sensitivity measure does not change more than 12 percent when it is calculated for a reasonable subset of initial test results (see Table I). If we use more than 8 environments it seems that average difference is less than 10 percent. We suppose that this behavior of measure is possible because the way environment reacts to tests is not dependent on other environments. This makes this sensitivity measure good candidate for fitness function in evolutionary programming.

B. Sensitivity measure as a fitness function

As stated before sensitivity measure can be used as fitness function so the next step was to check if proposed algorithm allows to quickly find best environments. Populations of sizes 8, 10 and 12 were tested. Each time algorithm was run 1000

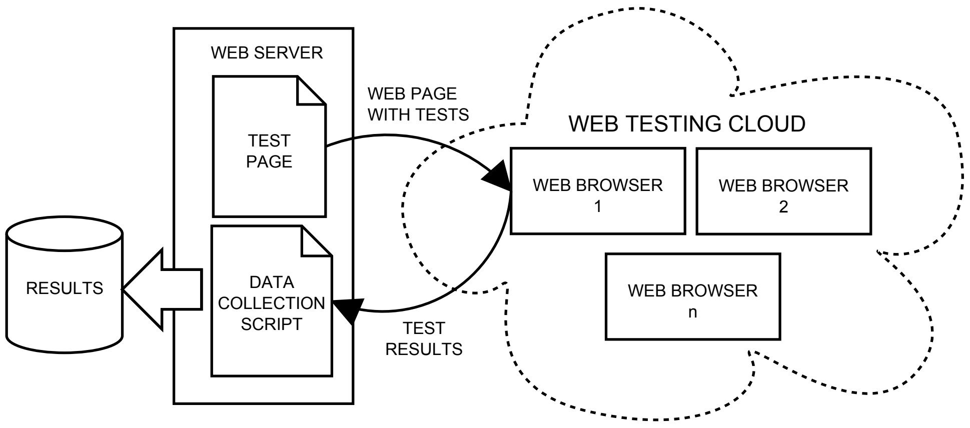

Fig. 2: Architecture of the system used for environment compatibility experiment.

TABLE I: Standard deviation of environment sensitivity calculated from random subsets from initial data.

	5 environments	8 environments	10 environments
100 random subsets	0.077	0.045	0.04
1000 random subsets	0.10	0.075	0.06
10000 random subsets	0.12	0.09	0.08

TABLE II: Averaged results after 1000 execution of environment selection algorithm. Difference is calculated against sensitivity calculated for all configurations together.

Population size	Max difference	Max iterations
n = 8, k = 4	15%	4.32 iterations
n = 10, k = 5	4.5%	4.09 iterations
n = 12, k = 6	3.8%	3.3 iterations

times and results were averaged. They are presented in Table II. For population size of 10 algorithm delivered a stable set of environments in less than 5 iterations.

In Table III we can see browsers selected by algorithm in more than 10% of cases along with their average sensitivity. Columns Operating system, Browser type and Browser Version define environment. Frequency shows percentage of times given configuration was chosen in top k results in 1000 runs of algorithm (eg. 75% means that is was chosen in 750 times). Average sensitivity is arithmetic mean of sensitivity value calculated for environment in all runs. If we compare this table with sensitivity calculated for all environments in Appendix A Table IV they basically match each other.

C. *Strategies for selecting environments for tests*

Results also proved common sense that better to test on older versions of software because newer versions have lot of compatibility problems already fixed. This can be seen on Figure 3 and 4 where sensitivity of environment is presented versus browser version. We had to normalize version numbering to $[0,1]$ because of different numbering schemes

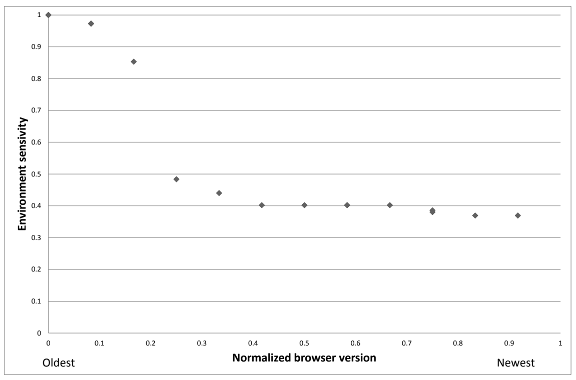

Fig. 3: Sensitivity of environment by browser version for Mozilla Firefox. Version numbers were normalized to be from 0 (oldest) to 1 (latest). You can see sudden improvement in HTML5 compatibility after third consecutive version.

used by browser vendors. We can consider several strategies to reduce number of environments used for tests. Simplest one is to establish a cut off point below that every environment is discarded. On Figure 3 we see that good cut off point will be sensitivity with value 0.5 because it clearly separates set. However other good strategy will be to discard those some environments that have similar sensitivity value. From Figure 3 and Table IV we see that Firefox from version 6 to 15 have sensitivity between 0.3 and 0.4. This means that we can choose one or several of them based on own preference (or random choice) because they behave more or less similarly during tests.

Other important aspect is that environment sensitivity can provide an order of tests. If we start with environments with highest sensitivity and some tests will fail, we can stop, fix defect and start over again. In our test case, testing complicated web pages on latest browser versions will likely be successful,

TABLE III: Average sensitivity for environments using proposed algorithm (averaged after 1000 runs) for configuration set of size 10. Frequency show how many times given environment was chosen by algorithm in top k best. Only those configurations with frequency more than 10% are shown.

Operating system	Browser type	Browser version	Frequency	Average sensitivity
LINUX	FIREFOX	2.0.0.17	77%	0.960
LINUX	KONQUEROR	4.8	77%	0.889
MAC OS X	CAMINO2	2.1.2	76%	0.802
LINUX	FIREFOX	1.5.0.12	76%	0.983
WINDOWS	FIREFOX	2.0.0.12	74%	0.963
MAC OS X	SAFARI	4.0.5	50%	0.766
WINDOWS	CHROME	3.0.182.2	31%	0.777
WINDOWS	CHROME	4.0.223.11	24%	0.609
WINDOWS	OPERA	10.00	16%	0.388
LINUX	FIREFOX	7.0.1	12%	0.323
LINUX	FIREFOX	6.0.1	10%	0.326

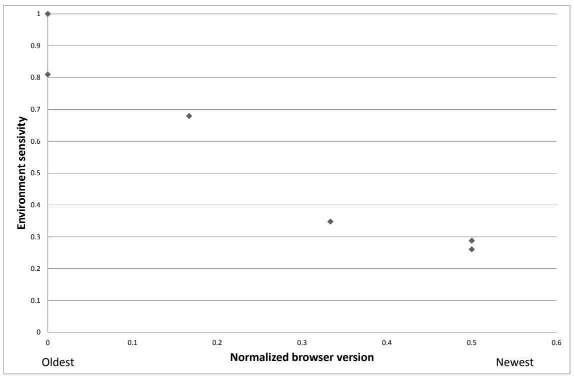

Fig. 4: Sensitivity of environment by browser version for Chrome. Version numbers were normalized to be from 0 (oldest) to 1 (latest). You can see improvement in HTML5 compatibility in latest versions. However it is not as steep as in Firefox browser.

because they are more HTML5 compatible. So better strategy will be to test on older versions and if they pass tests, then check on latest versions.

IV. Conclusions and future work

It seems that introduced environment sensitivity measure is a good way of measuring usefulness of environment for testing purposes. It provides analytical way to compare configurations and allows to use existing optimization techniques. For more complicated environments (that have several nodes in their CDG) we plan to use evolutionary algorithms. For presented browser testing case, cross-over and mutation operations were not feasible because they produced configurations that were not available in testing cloud. Introduction of environment domination (in Pareto sense) will allow to use existing methods used in multi-criteria optimization. Automated tests are usually run frequently in order to find out regression defects introduced during development. This causes tests to repeatedly oscillate between pass and fail states. We are now extending sensitivity model by introducing time line to take those changes into consideration and utilize historical information for more precise results.

We are also investigating possibility of using machine learning to correlate changes in application code base with historical test results to predict the best configuration and tests order to test on. This way when a new change is being introduced to software we can decide in which environment it should be tested in first place.

In our research we are planning to used multi-agent systems (See [5] and [6]) that will automatically deploy environments and optimize them for most efficient testing in terms of quality and resource consumption. Sensitivity is a useful measure to be used in algorithms that detect unusual behaviors like those mentioned in [7] and [8].

We are also considering introducing second measure based on probability that will cooperate with environment sensitivity that will allow us to better describe environment behavior and compare them in more than one category.

References

[1] L. Pobereznik, “Automatic generation and configuration of test environments,” in *Information Systems Architecture and Technology - Web Information Systems Engineering, Knowledge Discovery and Hybrid Computing*. Oficyna Wydawnicza Politechniki WrocÅĆawskiej, WrocÅĆaw, 2011, p. 303.

[2] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter, “Direct-dependency-based software compatibility testing,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE ’07. New York, NY, USA: ACM, 2007, pp. 409–412. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321696

[3] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter, “Effective and scalable software compatibility testing,” in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA ’08. New York, NY, USA: ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390640

[4] I. Yoon, “Compatibility testing for component-based systems,” Ph.D. dissertation, University of Maryland, 2010, hdl.handle.net/1903/11294.

[5] K. Cetnarowicz and V. Gruer, P. anf Hilare, “A formal specification of m-agent architecture,” in *Proc. Multi-Agent Systems CEEMAS 2001*, L. N. i. A. I. v. . S.-V. Keplicz B., Nawarecki E., Ed., Berlin, Heidelberg, 2002, pp. 62–72.
[6] K. Cetnarowicz, “From algorithm to agent,” in *Computational Science ICCS 2009, LNCS 5545 Springer Verlag*, 2009, pp. 825–834.
[7] K. Cetnarowicz and G. Rojek, “Behavior based detection of unfavorable resources,” in *Proc. Computational Science - ICCS 2004*, G. S. P. e. a. L. N. i. C. S. v. . S.-V. Bubak, M; VanAlbada, Ed., Berlin, Heidelberg, 2004, pp. 607–614.
[8] K. Cetnarowicz and G. Rojek, “Behavior evaluation with actions’ sampling in multi-agent system,” in *Proc. Multi-Agent Systems and Applications CEEMAS 2005*, P. V. L. N. i. C. S. v. . S.-V. Pechoucek, M; Petta, Ed., Berlin, Heidelberg, 2005, pp. 490–499.

Appendix

In this section we present a table with sensitivity values calculated for different versions of popular browsers running on various operating systems. Sensitivity was calculated at once based on all test results from all available configurations.. In production it is usually not possible to keep so many testing environments, so only a small subset of them is used for daily testing and more of them are added when needed (for example before product release). You can compare results from this table with values from Table III.

TABLE IV: Sensitivity values calculated for all configurations (only non-zero values are shown). In this case sensitivity was calculated for all environments at once.

System	Browser	Browser version	Sensitivity
LINUX	FIREFOX	1.5.0.12	1.000
LINUX	FIREFOX	2.0.0.17	0.971
WINDOWS	FIREFOX	2.0.0.12	0.971
LINUX	KONQUEROR	4.8	0.902
MAC OS X	CAMINO2	2.1.2	0.821
MAC OS X	SAFARI	4	0.202
WINDOWS	CHROME	3.0.182.2	0.798
WINDOWS	CHROME	4.0.223.11	0.659
WINDOWS	OPERA	10.00	0.445
LINUX	FIREFOX	7.0.1	0.405
LINUX	FIREFOX	6.0.1	0.405
MAC OS X	FIREFOX	11.0	0.364
MAC OS X	FIREFOX	12.0	0.364
MAC OS X	FIREFOX	13.0.1	0.364
MAC OS X	FIREFOX	14.0.1	0.364
MAC OS X	FIREFOX	15.0.1	0.358
WINDOWS	FIREFOX	11.0	0.358
WINDOWS	FIREFOX	16.0	0.341
LINUX	SAFARI	5.0	0.312
WINDOWS	CHROME	5.0.375.125	0.306
WINDOWS	CHROME	6.0.453.1	0.243
LINUX	CHROME	6.0.472.63	0.214
MAC OS X	SAFARI	6.0.1	0.208
WINDOWS	CHROME	7.0.517.44	0.173
LINUX	CHROME	20.0.1132.47	0.075
LINUX	CHROME	22.0.1229.94	0.052
MAC OS X	CHROME	22.0.1229.94	0.046
WINDOWS	SAFARI	5.0	0.046
WINDOWS	OPERA	11.64	0.017